



SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – II
FALL SEMESTER 2025-2026
ANSWER KEY
SLOT: A1 + TA1

Answer 1

a) Joins vs. Denormalization

Joins in document DBs: costly, not natively supported, require multiple lookups.

Denormalization: embedding course + grade info inside student profile - faster reads.

Trade-off: denormalization increases data redundancy but optimizes frequent read queries.

Efficiency: denormalization is more efficient in document DB context (optimized for read-heavy workloads like profiles).

b) Replication & Consistency in Multi-Campus Setup

Replication: ensures copies of student data across campuses.

Consistency levels:

Strong consistency - accurate but slower across campuses.

Eventual consistency - faster, but temporary stale reads possible.

Tunable consistency (quorum reads/writes) as middle ground.

Inter-campus synchronization issue: grade updates may lag leading to risk of conflict.

Solution: choose appropriate consistency level depending on query criticality (grades = strong consistency, analytics = eventual acceptable).

c) Partitioning Algorithm for Students' Data

Need: avoid query hotspots (too many requests hitting one shard).

Use hash-based partitioning on studentID for even distribution.

Avoid department-only partitioning - prevents hotspots.

Ensures balanced load, minimizes cross-shard queries.

Goal: minimize cross-shard queries + balance workload.

d) Indexing Strategies

i. Query: “students enrolled in course C101”

Create **secondary index on courseID** within enrollment array.

Use **multikey index** (since multiple courses per student).

Speeds up membership queries.

ii. Query: “average GPA of students in Computer Science department”

Index on department field (faculty/major).

Support aggregation pipeline for GPA averaging.

Possibly use compound index (department, GPA) for faster filtering + aggregation.



SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – II
FALL SEMESTER 2025-2026
ANSWER KEY
SLOT: A1 + TA1

Answer 2:

Model design

Document shape (embed for single-read):

Patient { patientId, name, demographics, visits: [{ visitId, date, doctor, notes }], prescriptions: [{ rxId, medication, startDate, endDate, dose }] } — embedding gives one-read retrieval of a complete patient record.

Indexing-ready field: keep prescriptions as an array of small subdocuments (medication + date fields normalized) to enable multikey indexing.

Scalability (when prescriptions grow large): if prescription history becomes very large, move prescriptions to a separate collection Prescriptions(patientId, medication, date, rxMeta) while keeping a compact embedded summary in Patient for single-read efficiency.

Challenges a traditional query processor

- **Collection scan / array scan:** it must examine every patient doc and every prescription subdocument — $O(N)$ expensive.
- **Unwinding arrays is costly:** the engine must expand arrays (unwind) to filter by medication & date before grouping/returning patientIds.
- **Network & I/O heavy in distributed setup:** scanning large documents across shards causes huge I/O and cross-node traffic.
- **Poor latency for interactive queries / analytics.**

Recommended indexing approach

Primary solution:

Create a **multikey compound index** on prescriptions.medication + prescriptions.startDate (or prescriptions.date). The DB uses the index to locate only prescription entries with medication = "Amoxicillin" and then does a range scan on the date field — avoids full-doc scan and array unwinding for most rows.

Alternative:

Maintain a small **inverted index collection** PrescriptionIndex { medication, date, patientId } (one document per prescription or per (patientId, medication, day)). Querying the index returns patientIds directly; then fetch full patient docs if needed. Single-index lookup + cheap date-range scan on compact index, minimal I/O before joining to patient docs.



SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – II
FALL SEMESTER 2025-2026
ANSWER KEY
SLOT: A1 + TA1

Distributed consistency challenges for updating a patient record

- Need for strong update visibility
- Trade-offs (CAP)
- Write protocol choices & issues
- Conflict handling
- Large embedded-doc updates

Answer 3

a) Choose: Column-family DB (Cassandra)

justification

- **Match to workload:** Cassandra excels at very high write throughput, linear horizontal scalability and low-latency reads/writes across many data centers — fits “millions of transactions daily” and geo distribution.
- **Data modeling fit:** wide-row/time-series model maps naturally to order histories and transaction streams (partition by userId, clustering by timestamp) for efficient append and range scans.
- **Replication strategy:** use datacenter-aware replication (replication factor) so each region serves local reads/writes while maintaining copies elsewhere; enables fast local reads and disaster tolerance.
- **Query processor behavior:** Cassandra requires query-driven schema design (model tables for known access patterns) which yields predictable fast queries when designed correctly; avoids expensive ad-hoc joins.
- **Distributed consistency:** tunable consistency (ONE/QUORUM/LOCAL_QUORUM) lets architects trade latency vs correctness — choose LOCAL_QUORUM for cross-DC safety with acceptable latency.
- **Operational advantage:** predictable performance at scale, built-in failure tolerance and simple operational scaling compared to heavy cross-node joins in document DBs.

b) Normalization vs Denormalization & indexing

- **Recommendation: Denormalization** for “user purchase history” and “top trending products” to achieve low-latency reads at scale.
- **How to denormalize:**
 - Store per-user order rows partitioned by userId (append order events there) for direct fast retrieval.



SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – II
FALL SEMESTER 2025-2026
ANSWER KEY
SLOT: A1 + TA1

- Maintain a compact “user_recent_purchases” table or cache for most recent N purchases.
- Maintain a separate, update-friendly counter/table for product trending (time-windowed counters or rolling aggregates).
- **Indexing influence (Cassandra specifics):**
 - Rely primarily on **primary key + clustering columns** for fast access (design partition & clustering to match queries).
 - **Avoid secondary indexes** for high-cardinality transactional data; they are inefficient and can cause coordinator bottlenecks.

Answer 4

a.

```
CREATE TABLE publications_by_author (  
  author_id text,  
  year int,  
  paper_id text,  
  author_name text,  
  conference text,  
  title text,  
  PRIMARY KEY (author_id, year, paper_id)  
) WITH CLUSTERING ORDER BY (year DESC, paper_id DESC);
```

b.

```
INSERT INTO publications_by_author (  
  author_id, year, paper_id, author_name, conference, title  
) VALUES (  
  'A101', 2024, 'P5001', 'Dr. Rao', 'VLDB', 'Scalable Partitioning Algorithms'  
)
```

c.

```
CREATE TABLE authors_by_conf_year (  
  conference text,  
  year int,  
  paper_id text,  
  author_id text,  
  author_name text,  
  PRIMARY KEY ((conference, year), paper_id)  
)
```



SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – II
FALL SEMESTER 2025-2026
ANSWER KEY
SLOT: A1 + TA1

SELECT author_id, author_name FROM authors_by_conf_year
WHERE conference='VLDB' AND year=2024;

d.

DELETE FROM publications_by_author
WHERE author_id='A101' AND year < 2020;

e.

SELECT * FROM publications_by_author
WHERE author_id='A101' LIMIT 1;

Answer 5

a. Expected number of gossip messages/sec

- Each of the 12 nodes sends 1 gossip message/sec → total = $12 \times 1 = 12$ messages/sec.
- (Pairwise/bidirectional pairing yields same expected total: 6 pairs $\times$ 2 directions = 12 messages/sec.)

b. Network overhead from gossip (steady state)

- Total bytes/sec = 12 messages/sec $\times$ 400 bytes = 4800 bytes/sec.
- In KB/s (1024 bytes = 1 KB): $4800 \div 1024 \approx 4.6875$ KB/s ($\approx$ **4.69 KB/s** cluster-wide).
- Per-node traffic: send = 400 B/s, recv $\approx$ 400 B/s → per-node $\approx$ 800 B/s $\approx$ **0.78 KB/s**.

c. Effect when 3 nodes go offline (gossip + commit log + hinted handoff)

- **Gossip:** remaining nodes detect missed nodes and mark the 3 nodes as UNREACHABLE; status propagated so coordinators avoid routing reads/writes to them.
- **Commit log:** coordinators still durably write incoming chat messages to local commit log before ack; messages are not lost even if some replicas are down.
- **Hinted handoff:** coordinators that attempted writes to the unreachable replicas keep *hints* (info about missed writes). When offline nodes come back, hints are replayed to apply missed writes so those nodes converge.
- **Net effect:** clients get durable acknowledgments; replication to offline nodes is deferred and completed later via hinted handoff, but there is a window during which those three replicas are stale.

d. Source of inconsistent chat delivery order & mitigations

- **Origin:** asynchronous replication + network delays + independent local commit ordering cause different replicas to observe writes in different orders, so read-from-



SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – II
FALL SEMESTER 2025-2026
ANSWER KEY
SLOT: A1 + TA1

different-replicas can show different orders. Commit log guarantees durability but not global ordering.

- **Mitigations:**

- Increase write/read consistency (e.g., QUORUM or LOCAL_QUORUM) so most clients read from a majority that agrees on order.
- Use a single-partition key for a chat stream so clustering order (by timestamp/sequence) is deterministic.
- Attach **client-assigned sequence numbers** or logical timestamps (Lamport) to messages and enforce ordering on reads.
- For strict ordering, use synchronous replication or a leader/coordinator per partition (trades latency for ordering guarantees).

e. Schema change propagation & one risk of lag

- **Propagation:** nodes advertise their schema version via gossip; when a node sees a different schema version it requests the new schema from peers and installs it — gossip circulates the schema metadata until all 12 nodes converge.
- **Risk if propagation lags:** a node with an old schema may accept or execute requests incorrectly (write failures, query errors, or inconsistent storage layout), causing schema disagreement errors or client-visible inconsistencies until propagation completes.